Journal of Computer and System Sciences 78 (2012) 119–141

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

An action-based approach to the formal specification and automatic analysis of business processes under authorization constraints

Alessandro Armando [a,b], Enrico Giunchiglia [a,*], Marco Maratea [a], Serena Elisa Ponta [a,c]

[a] DIST, University of Genova, Genova, Italy
[b] Security & Trust Unit, Fondazione Bruno Kessler, Trento, Italy
[c] SAP Research Sophia-Antipolis, Mougins, France

ARTICLE INFO

Article history:
Received 9 September 2009
Received in revised form 1 July 2010
Accepted 10 February 2011
Available online 15 March 2011

Keywords:
Knowledge representation and reasoning
Action languages
Business processes

ABSTRACT

Business processes under authorization control are sets of coordinated activities subject to a security policy stating which agent can access which resource. Their behavior is difficult to predict due to the complex and unexpected interleaving of different execution flows within the process. Serious flaws may thus go undetected and manifest themselves only after deployment. For this reason, business processes are being considered a new, promising application domain for formal methods and model checking techniques in particular. In this paper we show that action-based languages provide a rich and natural framework for the formal specification of and automated reasoning about business processes under authorization constraints. We do this by discussing the application of the action language $\mathcal{C}$ to the specification of a business process from the banking domain that is representative of an important class of business processes of practical relevance. Furthermore we show that a number of reasoning tasks that arise in this context (namely checking whether the control flow together with the security policy meets the expected security properties, building a security policy for the given business process under given security requirements, and finding an allocation of tasks to agents that guarantees the completion of the business process) can be carried out automatically using the Causal Calculator CCALC. We also compare $\mathcal{C}$ with the prominent specification language used in model-checking.[1]

1. Introduction

Business processes under authorization control are sets of coordinated activities subject to a security policy stating which agent can access which resource. The order according to which the activities must be executed is given by the workflow, which is usually specified by means of graphical notations, e.g., Petri nets [2], Business Process Modelling Notation (BPMN) [3]. The specification of the policy is usually given in terms of a basic access control model (e.g., the Role-Based Access Control (RBAC) model [4]) possibly enriched with features providing the flexibility required by the application domain (e.g., delegation) and mechanisms that are necessary to meet mandatory regulations (e.g., separation of duty constraints). The behavior of business processes is difficult to predict due to the complex and unexpected interleaving of different execution flows within the process. Serious flaws (either in the control flow or in the security policy, or in both) may thus go

* Corresponding author at: Viale F. Causa 15, 16145, Genova, Italy. Fax: +390103532948.
E-mail addresses: armando@dist.unige.it (A. Armando), enrico@dist.unige.it (E. Giunchiglia), marco@dist.unige.it (M. Maratea), serena.ponta@unige.it (S.E. Ponta).

[1] This paper is an extended and thoroughly revised version of Armando et al. (2009) [1].

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2011.02.015

undetected and manifest themselves only after deployment. Since these undesirable behaviors are very difficult to spot by simple inspection of the system, or by simulation, business processes are being considered a new, promising application domain for formal methods and model checking techniques in particular. In previous works, e.g., [5,6], it has been shown that model checking can be profitably used for the automatic analysis of business processes, and a number of techniques have been designed to alleviate the state explosion problem, see, e.g., [7]. Yet, their applicability to business processes appears to be problematic as state-of-the-art model checkers—being geared to the analysis of hardware designs—require the system to be modeled as the composition of independent (yet interacting) sub-components. On the contrary, business processes and the associated security policy are best viewed as a collection of actions subject to a given workflow pattern and a set of independent access control rules.

Reasoning about actions and change is a long standing research area in Artificial Intelligence (AI), and several languages for knowledge representation [8] (e.g., logic programming [9,10] and action languages [11–13]) and automated reasoning tools (e.g., SMODELS [14], DLV [15], DLV$^{\mathcal{K}}$ [16], CCALC [17]) have been put forward. In this paper we show that the action-based language $\mathcal{C}$ [18] provides a rich and natural specification framework for supporting formal declarative specifications of business processes under authorization constraints and that a number of reasoning tasks that arise in this context can be automated by using the Causal Calculator CCALC. The effectiveness of the proposed approach is illustrated by using $\mathcal{C}$ to specify the Loan Origination Process (LOP), a business process from the banking domain that is representative of an important class of business processes of practical relevance as it features many aspects that frequently occur in practice: non-trivial interplay between the control flow and the security policy, sophisticated access-control policies, events and tasks with nondeterministic, conditional and indirect effects. To the best of our knowledge, no other approach to the specification and automatic analysis of business processes encompasses all the above aspects: [19,20] consider complex security policy but do not take into account the workflow; [21,22] analyze SoD constraints while considering both the workflow and the security policy but do not support implicit preconditions of tasks, events, roles hierarchy, and delegation; finally [5] considers the workflow and an RBAC security policy enhanced with delegation but does not consider, e.g., nondeterministic, indirect, and conditional effects of tasks, events, and policy exceptions.

More specifically we show that $\mathcal{C}$ supports

- the separate specification of the workflow and of the associated security policy;
- the formal and declarative specification of a wide range of security policies;
- the specification of a variety of business process features, e.g., events, tasks with nondeterministic, indirect, and conditional effects; and, most importantly,
- the seamless integration of all the above aspects.

Moreover $\mathcal{C}$ provides modelers with the ability to specify the system incrementally. This is an important feature that is seldom supported by the specification languages of state-of-the-art model-checkers. We substantiate this observation by comparing $\mathcal{C}$ with SMV, the specification language of the NUSMV model checker [23,24].

We also show that a variety of automated reasoning tasks occurring in the domain of business processes can be recast as (a sequence of) satisfiability problems of $\mathcal{C}$ specifications which can be automatically tackled with the aid of CCALC:

1. to establish whether the control flow together with the security policy meets the expected security properties;
2. for a given number of agents, synthesize a security policy for the business process under given security requirements;
3. for a given number of agents and for all execution flows, find (if any) an assignment of activities to agents ensuring the process executability according to the given security policy.

The rest of the paper is structured as follows. In the next section we provide a brief introduction to business processes under authorization constraints and describe our working example, which will be used throughout the paper. In Section 3 we overview the action language $\mathcal{C}$. In Section 4 we show how a business process under authorization constraints can be modeled in $\mathcal{C}$. In Section 5 we describe the analysis of the LOP carried out with CCALC. Then, in Section 6 we compare $\mathcal{C}$ and SMV. In Section 7 we discuss the related work and we conclude in Section 8 with some final remarks.

2. Business processes under authorization constraints

Let us consider the Loan Origination Process (LOP) graphically presented in Fig. 1. The workflow of the process is represented by means of an extended elementary net (see, e.g., [25]), a simple Petri net [2], extended with conditional arcs between places and transitions.

Let a *fact* be an atomic proposition. A *literal* is either a fact or its negation.

Formally, an *extended elementary net* is a 6-uple $\langle S, T, A, F, \gamma, M_0\rangle$ where

- S and T are finite sets of *places* and *transitions*, respectively, such that $S \cap T = \emptyset$;
- $A \subseteq (S \times T) \cup (T \times S)$ is the *flow relation*, representing a set of directed *arcs* connecting places and transitions. Places from which arcs run to a transition are called the *input places* of the transition; places to which arcs run from a transition are called the *output places* of the transition;

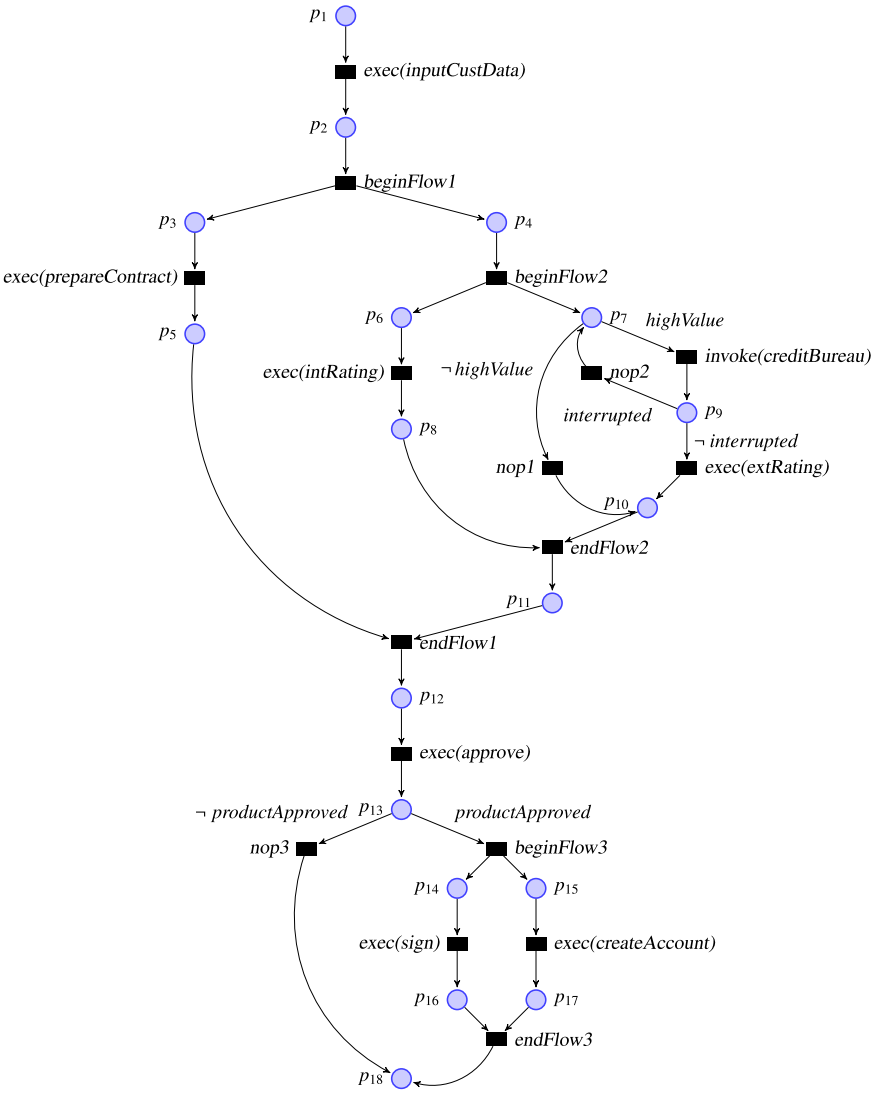

Fig. 1. Extended elementary net for the LOP.

- F is a set of facts, and the set of literals over F is denoted with $\mathcal{L}$. A set of literals $\mathcal{L}$ is consistent if and only if $\mathcal{L}$ does not contain a fact and its negation;
- $\gamma : (S \times T) \to \mathcal{L}$ is a function that associates arcs between places and transitions with literals over F expressing *applicability conditions*;
- $M_0 \subseteq S$ defines the *initial marking* for places.

In Fig. 1

- $S = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16}, p_{17}, p_{18}\}$;
- $T = \{exec(inputCustData), exec(prepareContract), exec(intRating),$
  $exec(extRating), invoke(creditBureau), exec(approve), exec(sign),$
  $exec(createAccount), beginFlow1, beginFlow2, beginFlow3,$
  $endFlow1, endFlow2, endFlow3, nop1, nop2, nop3\}$;
- $A = \{(p_1, exec(inputCustData)), (p_2, beginFlow1),$
  $(p_3, exec(prepareContract)), (p_4, beginFlow2),$
  $(p_5, endFlow1), (p_6, exec(intRating)),$
  $(p_7, invoke(creditBureau)), (p_7, nop1),$
  $(p_8, endFlow2), (p_9, exec(extRating)),$
  $(p_9, nop2), (p_{10}, endFlow2),$

$(p_{11}, endFlow1)$, $(p_{12}, exec(approve))$,
$(p_{13}, nop3)$, $(p_{13}, beginFlow3)$,
$(p_{14}, exec(sign))$, $(p_{15}, exec(createAccount))$,
$(p_{16}, endFlow3)$, $(p_{17}, endFlow3)$,
$(exec(inputCustData), p_2)$, $(beginFlow1, p_3)$,
$(beginFlow1, p_4)$, $(exec(prepareContract), p_5)$,
$(beginFlow2, p_6)$, $(beginFlow2, p_7)$,
$(exec(intRating), p_8)$, $(nop1, p_{10})$,
$(invoke(creditBureau), p_9)$, $(nop2, p_7)$,
$(exec(extRating), p_{10})$, $(endFlow2, p_{11})$,
$(endFlow1, p_{12})$, $(exec(approve), p_{13})$,
$(nop3, p_{18})$, $(beginFlow3, p_{14})$,
$(beginFlow3, p_{15})$, $(exec(sign), p_{16})$,
$(exec(createAccount), p_{17})$, $(endFlow3, p_{18})\}$;

- $F = \{highValue, interrupted, productApproved\}$;
- $\gamma(p_7, invoke(creditBureau)) = highValue$, $\gamma(p_7, nop1) = \neg highValue$,
  $\gamma(p_9, nop2) = interrupted$, $\gamma(p_9, exec(extRating)) = \neg interrupted$,
  $\gamma(p_{13}, beginFlow3) = productApproved$, $\gamma(p_{13}, nop3) = \neg productApproved$;
- $M_0 = \{p_1\}$.

A state of the extended elementary net is a pair (M, L), where M is a marking and $L \subseteq \mathcal{L}$ is a maximally consistent set of literals (i.e., a truth-value assignment to the facts in F). A *marking* represents an execution state of the extended elementary net, initially set to M_0. Given a marking M, a set of literal L and a place p, if $p \in M$ then we say that the place p *contains a token*, otherwise we say that p *is empty*. Starting from M_0, a transition of the extended elementary net can *fire* if there is a token in every input place of the transition and the conditions associated with the arcs between the input places and the transition hold in L. As a consequence, a new marking is reached where, for a transition that has fired, each output place contains a token, while the token of each input place is eliminated, and L is updated by the effects of the transition.

Fig. 1 represents the LOP as an extended elementary net. Notice that in our model we have some artificially added transitions to begin/end a flow of business activities (i.e., transitions *beginFlow1*, *beginFlow2*, *beginFlow3*, *endFlow1*, *endFlow2*, and *endFlow3*) and to skip certain operations according to particular conditions (i.e., transitions *nop1*, *nop2*, and *nop3*). Such transitions are thus inserted for workflow modeling purposes, and are triggered as soon as their preconditions hold.

In our example, the process starts with the input of the customer's data (*inputCustData*). Afterwards a contract for the current customer is prepared (*prepareContract*) while the customer's rating evaluation takes place concurrently. By means of the rating evaluation the bank establishes if the customer is suitable to receive the loan. In our model, the execution follows different paths: if the amount of the requested loan is not high ($\neg$*highValue*), then an internal rating suffices (*intRating*); otherwise, an external rating (*extRating*) is executed concurrently by invoking a Credit Bureau, a third-party financial institution. As soon as it is ascertained that the external rating is needed, a request to obtain the credit information about the current customer is sent to the Credit Bureau (*invoke*(*creditBureau*)). By using this information, the external rating evaluation is performed by executing the task *extRating*. Notice that the invocation of the Credit Bureau and the execution of the task *extRating* must be performed by the same agent. In case there is a forbidden access to the information sent by the Credit Bureau to the bank, i.e., an agent who is not the director and different from the one who sent the request has accessed the information exchanged, then the execution of the task is prevented and the rating evaluation is interrupted. In case of interruption, the director of the bank must re-invoke the Credit Bureau and execute the task *extRating*. Thus, the loop in Fig. 1 can be executed at most once.[2] The loan request must then be approved (*approve*) by the bank. As soon as the customer and the bank have reached an agreement, the contract is signed (*sign*) and an account for the customer is created concurrently (*createAccount*). Notice that the execution of a task may affect the state of the process. In particular

- task *inputCustData* may modify the state of the execution by issuing statements about the type of customer (i.e., if it is industrial, *isIndustrial*) and the amount of the loan (i.e., if it is high, *highValue*),
- task *intRating* may issue statements about the evaluation of the customer, i.e., if the internal rating of the customer is positive, and
- tasks *approve* may issue a statement asserting if the proposed product is suitable or not for the customer.

Moreover a task may perform further operations affecting the state of the process only if some conditions hold. In particular, in case the customer is industrial, during the evaluation of the internal rating the bank also establishes if it is important to deserve a particular care to the customer, i.e., if the customer is industrial the task *intRating* may issue a statement asserting if the customer has a high profile (*highProfileIndCust*). Finally, different statements may determine new process features, e.g.,

[2] Note that this does not limit the specification capabilities of our approach: the fact that such loop can be executed at most once is, by chance, the result of the separate specification of the workflow (which allows for multiple executions of the loop) with a *particular* security policy adopted.

Table 1
Permission assignment for the LOP.

Task	Role
inputCustData	*preprocessor*
prepareContract	*postprocessor*
intRating	if (*isIndustrial*) then *postprocessor* else *preprocessor*
extRating	if (*interrupted*) then *director* else *supervisor*
approve	if (*lowRisk*) then *manager* else *director*
sign	if (*highProfileIndCust*) then *director* else *manager*
createAccount	*postprocessor*

the *lowRisk* condition is used to denote a situation in which the internal rating is positive and the amount of the loan is not high.

An agent can execute a task only if she has the required permissions. As it is common in the business domain, the security policy of the LOP relies on an access control model based on RBAC enhanced with delegations and separation of duty constraints. According to the RBAC model [4], in order to perform a task an agent must be assigned a role enabled to execute the task and the agent must be also active in that role. The roles used in our case study are *director*, *manager*, *supervisor*, *postprocessor*, and *preprocessor*. Roles can be organized hierarchically. In our case study, a *director* is more senior than a *manager* and a *supervisor* is more senior than a *postprocessor*. Senior roles inherit the permissions to perform tasks assigned to more junior roles. Thus, an agent can execute a task if her role

- is directly assigned the required permissions; or
- is more senior than a role owning such permissions.

In our case study we consider permission assignments subject to the following requirements:

- task *inputCustData* is assigned to role *preprocessor*;
- tasks *prepareContract* and *intRating* cannot be assigned to roles *director* or *manager*;
- task *createAccount* cannot be assigned to roles *director*, *manager*, or *preprocessor*;
- tasks *approve* and *sign* cannot be assigned to roles *preprocessor*, *postprocessor*, or *supervisor*;
- if a customer is industrial, then role *preprocessor* cannot be enabled to perform the task *intRating*;
- if the process has been interrupted, the task *extRating* has to be performed by a *director*;
- if the process has not been interrupted, the task *extRating* cannot be assigned to roles *director*, or *manager*;
- if the risk of the loan is not low, then role *manager* cannot be enabled to perform the task *approve*;
- if the industrial customer has a high profile, then role *manager* cannot be enabled to perform the task *sign*.

Table 1 shows a possible permission assignment for the LOP satisfying the requirements presented above. Notice that the invocation of the Credit Bureau does not appear in Table 1 as this activity has to be executed by the same agent of the task *extRating* and, as a consequence, uses its permission assignment.

A *user assignment* is a relation that associates agents and roles. We consider a static assignment of agents to roles subject to the following requirements:

- there must be only one director,
- the director must not be assigned to any other role,
- an agent must not be assigned to roles hierarchically related, e.g., an agent cannot be assigned to the roles *supervisor* and *postprocessor*, and
- an agent can be assigned to two different roles at most.

Notice that the requirements we have specified above, for both permission and user assignments, correspond to a *class* of security policies, given, e.g., the use of conditional statements in the permission assignment.

A possible user assignment for the LOP is given by the following assignments of agents to roles: *davide*, the director, *maria* and *marco*, managers, *pierPaolo*, who can act both as preprocessing clerk and as postprocessing clerk, *pierSilvio*, who can act both as preprocessing clerk and as supervisor, *pietro*, postprocessing clerk, and *stefano*, supervisor.

Our RBAC access control model is enhanced with delegation that represents a typical flexibility requirement. Following the idea of conditional delegation presented in [26], we consider *delegation rules* of the form

$$\langle \mathit{PreConds}, \mathit{ARole}, \mathit{DRole}, \mathit{Task} \rangle$$

where *ARole* and *DRole* are roles, *Task* is a task, and *PreConds* is a set of conditions that must hold for the delegation to be applicable. A delegation rule states that if *PreConds* holds and *ARole* is authorized to perform *Task* according to the permission assignments, then *ARole* can delegate *DRole* to execute *Task*. We also distinguish between *grant* and *transfer* delegation [27]. Considering a grant delegation rule, both agents involved are then allowed to perform the task being

Table 2
Delegation rules of the LOP.

Name	Type	Delegation rule
D1	transfer	$\langle \neg isIndustrial, supervisor, postprocessor, extRating \rangle$
D2	grant	$\langle intRatingPositive, manager, supervisor, approve \rangle$
D3	grant	$\langle \neg highValue, director, supervisor, sign \rangle$

Table 3
Critical tasks of each object-based SoD of the LOP.

Name	Object	Critical tasks
C1	customer's data	*inputCustData*, *prepareContract*, *intRating*, *extRating*
C2	rating report	*intRating*, *extRating*
C3	contract	*prepareContract*, *approve*, *sign*

delegated; considering a transfer delegation rule the ability to perform the task is transferred and the agent who executed the delegation cannot perform the delegated task anymore. Notice that our delegation rules express task delegation rather than role delegation. In fact, the delegated agent does not acquire a new role but she only obtains the permission to perform *Task* by means of *ARole*. Also notice that in [26] three different kinds of conditions are considered, i.e., temporal, value, and workload delegation conditions, however in our work *PreConds* can only be related to the value of the attributes of the process, i.e., value delegation conditions. In our case study we consider the delegation rules in Table 2.

We also consider the ability of the director to disable an agent from performing a task overriding the security policy in use. As a result, an agent can execute a task if she is granted the permission by means of the RBAC model or by means of delegation unless the director explicitly disables her from performing the task.

Finally, the RBAC model of our case study is enhanced with a mechanism that is necessary to satisfy separation of duty (SoD) constraints. SoD constraints are used for internal control and amount to requiring that some critical tasks are executed by different agents (see [5] for a survey on SoD). In this paper we focus on a relaxed form of *object-based SoD* according to which an agent can access the same object through different roles as long as she does not perform all the tasks accessing that object. For each object involved in the LOP, we define the corresponding critical tasks consisting of all and only the tasks accessing the object. We then assume that an agent cannot execute all the critical tasks associated to each object. Such associations are presented in Table 3.

3. The action language $\mathcal{C}$

Action languages are high level formalisms for expressing *actions* and how they affect the world described with a set of atomic formulas called *fluents*. Thus, the signature σ of the language is partitioned into the *fluent symbols* σ^{fl} and the *action symbols* σ^{act}. Intuitively, *actions* are a subset of the interpretations of σ^{act} while *states* are a subset of the interpretations of σ^{fl}. A formula in σ is a propositional combination of atoms. Each action language differs from the others for the constructs used to characterize how actions affect states.

$\mathcal{C}$ [18] is an expressive propositional action language allowing for two kinds of propositions: *static laws* of the form

$$\textbf{caused } F \textbf{ if } G \tag{1}$$

and *dynamic laws* of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H, \tag{2}$$

where F and G are *fluent formulas* (i.e., formulas in σ^{fl}) and H is an *action formula* (i.e., a formula in σ). In a proposition of either kind, the formula F will be called the *head* of the law.

An *action description* is a set of propositions. Consider an action description D. A *state* is an interpretation of σ^{fl} that satisfies $G \supset F$ for every static law (1) in D. A *transition* is any triple $\langle s, a, s' \rangle$ where s, s' are states and a is an action; s is the *initial* state of the transition, and s' is its *resulting* state. A formula F is *caused* in a transition $\langle s, a, s' \rangle$ if it is

- the head of a static law (1) from D such that s' satisfies G, or
- the head of a dynamic law (2) from D such that s' satisfies G and $s \cup a$ satisfies H.

A transition $\langle s, a, s' \rangle$ is *causally explained* according to D if its resulting state s' is the only interpretation of σ^{fl} that satisfies all formulas caused in this transition.

The *transition diagram* represented by an action description D is the directed graph which has the states of D as nodes and includes an edge from s to s' labeled a for every transition $\langle s, a, s' \rangle$ that is causally explained according to D. Intuitively, if $\langle s, a, s' \rangle$ is a transition of the diagram, the concurrent execution of the atomic actions satisfied by a causes a transition from the state s to the state s'. Despite the fact that $\mathcal{C}$ consists of only two kinds of propositions, several other propositions

Table 4
Abbreviations for causal laws.

Abbreviation	Expanded form	Informal meaning
nonexecutable H' **if** F	**caused** $\bot$ **after** $H' \wedge F$	$\neg F$ is a precondition of H'
H' **causes** F **if** G	**caused** F **if** $\top$ **after** $G \wedge H'$	F is true after H' is executed in a state in which G is true
H' **may cause** F **if** G	**caused** F **if** F **after** $H' \wedge G$	F is true by default after H' is executed in a state in which G is true
default F	**caused** F **if** F	F is true by default
caused F **if** G **unless** Q	**caused** F **if** $G \wedge \neg Q$, **default** $\neg Q$	F is true after G in a state in which Q is false
constraint F	**caused** $\bot$ **if** $\neg F$	F must be true
inertial F	**caused** F **if** F **after** F, **caused** $\neg F$ **if** $\neg F$ **after** $\neg F$	F is inertial
exogenous F	**default** F, **default** $\neg F$	F is exogenous

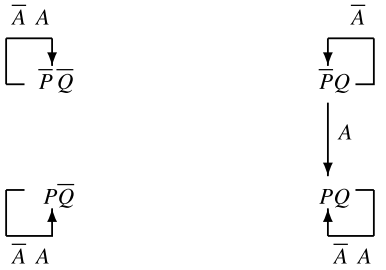

Fig. 2. Transition diagram for action description (3).

can be defined as abbreviations of either (1) or (2), modeling, e.g., actions' preconditions, actions' nondeterministic effects, fluents with default values, and inertial fluents. The abbreviations used in this paper are given in Table 4, where F and G are defined as before, and H' is a formula in σ^{act}.

Example. (See [18].) Let $\sigma^{fl} = \{P, Q\}$, $\sigma^{act} = \{A\}$, and let D consist of the propositions

inertial P, Q,

caused P **after** $Q \wedge A$. (3)

The second line of (3) tells us that P is made true by the execution of A if the precondition Q is satisfied (as, for instance, in the familiar shooting example which corresponds to *Dead* as P, *Loaded* as Q, and *Shoot* as A). Preconditions are specific conditions holding in a state they are executed. According to Table 4, (3) is a shorthand for

caused P **if** P **after** P,

caused $\neg P$ **if** $\neg P$ **after** $\neg P$,

caused Q **if** Q **after** Q,

caused $\neg Q$ **if** $\neg Q$ **after** $\neg Q$,

caused P **if** *True* **after** $Q \wedge A$. (4)

In this action description, there are 4 states (PQ, $P\overline{Q}$, $\overline{P}Q$, $\overline{P}\,\overline{Q}$)[3] and 2 actions ($A$ and $\overline{A}$). Consequently, there are $4 \times 2 \times 4 = 32$ transitions. Out of these, 8 transitions are causally explained: $\langle \overline{P}Q, A, PQ \rangle$ and the transitions of the form $\langle s, a, s \rangle$ where $s \neq \overline{P}Q$ or $a \neq A$.

Fig. 2 shows the corresponding transition diagram. We can see from it that each of the actions A, $\overline{A}$ can be executed in any state in exactly one way. To check that the transition $\langle \overline{P}Q, A, PQ \rangle$ is causally explained, note that the formulas

[3] We represent a propositional interpretation by listing the literals that are satisfied by it. $\bar{L}$ is the literal complementary to L.

Table 5
Fluents and their informal meaning.

Fluent	Meaning
$activated(a, r)$	agent a is playing role r
$accessed(a)$	agent a has accessed the information sent by the Credit Bureau
$delegated(a, r, t)$	agent a is delegated by an agent in role r to perform task t
$disabled(a, t)$	agent a cannot perform task t
$pa(r, t)$	role r has the permission to perform task t
$ua(a, r)$	agent a is assigned to role r
$granted(a, r, t)$	agent a obtained by means of role r the permission to perform task t
$executed(a, t)$	agent a has executed task t
$invoked(a, r, e)$	agent a has invoked entity e obtaining the authorization from role r
$senior(r_1, r_2)$	role r_1 is more senior than or is as senior as r_2
lowRisk	the risk associated with the loan is low
highValue	the loan amount is high w.r.t. the financial status of the customer
isIndustrial	the customer is industrial
highProfileIndCust	the industrial customer has a high profile
intRatingPositive	the customer's internal rating is positive
productApproved	both the customer and the bank agree on the contract
interrupted	the execution of the process is interrupted
$p_1, \ldots, p_{18}$	the places of the extended elementary net (cf. Fig. 1)

Table 6
Actions and their informal meaning.

Actions	Meaning
$exec(a, r, t)$	agent a executes task t by means of role r
$invoke(a, r, e)$	agent a invokes entity e obtaining the authorization from role r
$disable(a1, a2, t)$	agent $a1$ disables agent $a2$ from performing task t
$d1(a1, a2)$	agent $a1$ delegates agent $a2$ by means of delegation rule *d1*
$d2(a1, a2)$	agent $a1$ delegates agent $a2$ by means of delegation rule *d2*
$d3(a1, a2)$	agent $a1$ delegates agent $a2$ by means of delegation rule *d3*
beginFlow1	dummy activity to begin the first flow of business activities in Fig. 1
endFlow1	dummy activity to end the first flow of business activities in Fig. 1
beginFlow2	dummy activity to begin the second flow of business activities in Fig. 1
endFlow2	dummy activity to end the second flow of business activities in Fig. 1
nop1, *nop2*, *nop3*	dummy activities to skip operations

caused in this transition are the heads Q, P of the 3rd and 5th propositions in (4), and that the resulting state PQ of the transition is the only interpretation that satisfies both heads. We can also see that $\langle \overline{P}Q, \overline{A}, PQ \rangle$ is not causally explained: the only formula caused in this transition is the head Q of the 3rd proposition in (4), and Q is satisfied by more than one interpretation.

4. Formal modeling of business processes with $\mathcal{C}$

We now present a list of desired features for modeling business processes under authorization constraints, and we show the natural correspondence between the features, exemplified by using the LOP as a working example, and the $\mathcal{C}$ constructs we use. This section clearly shows that $\mathcal{C}$ is a well-suited modeling language for our domain.

The set of fluents involved consists of the fluents listed in Table 5. All the fluents in Table 5 but $pa(r, t)$, *lowRisk*, $granted(a, r, t)$, $activated(a, r)$, $accessed(a)$, and $disabled(a, t)$ are inertial. The set of actions involved is listed in Table 6. All the fluents in Table 5 and actions in Table 6 have to be suitably instantiated for all tasks and roles in Table 1 and the set of available agents.

4.1. Internal and external events

Events are effects of activities which are not explicitly represented in the model. We distinguish between internal and external events

- internal events are effects of procedures implemented within the process but not explicitly represented in the model;
- external events are effects of activities which are not part of the process but whose effects have an impact on the process itself.

Examples in the LOP are, respectively,

- the activation of a role, which is modeled as an internal event making sure that an agent is active in a role only if she is assigned to that role; and
- the access to some information by an agent not allowed to, which is modeled as an external event making sure that once an agent has accessed the information, a trace of this event remains in the process.

Internal and external events are modeled in $\mathcal{C}$ as exogenous fluents. Exogenous fluents are fluents that can change their value during the transition from one state to another (unless there is some other law constraining their values). In the case of the LOP, two exogenous fluents are used:

exogenous $activated(a, r), accessed(a)$.

The former is subject to the law

caused $\neg activated(a, r)$ **if** $\neg ua(a, r)$

to state that an agent cannot be active in a role she is not assigned to. The latter is subject to the law

caused $accessed(a)$ **after** $accessed(a)$

to state that the access to the information exchanged in the process by an agent is not reversible.

4.2. Execution of tasks and invocation of entities

The execution of a task can take place only if its preconditions are met. Preconditions of each task contain information about

- the input places of the corresponding transition in the extended elementary net;
- the condition associated with the arc from the input places, if any; and
- the authorized agent that will perform the task by means of the fluent $granted(a, r, t)$.

In $\mathcal{C}$, preconditions can be expressed by the law

nonexecutable H **if** $\neg F$ (5)

where H is an action formula and F is a formula expressing the preconditions of H. Consider a task t, and let I be the set of input places of the transition, and I' the set of conditions on the arcs from the input places (if any). Task preconditions have the following pattern:

$$\textbf{nonexecutable } exec(a, r, t) \textbf{ if } \neg\Bigg(\bigwedge_{p\in I} p \wedge \bigwedge_{f\in I'} f \wedge granted(a, r, t)\Bigg).$$

An example about *intRating* is the following

$$\textbf{nonexecutable } exec(a, r, intRating) \textbf{ if } \neg\big(p_6 \wedge granted(a, r, intRating)\big).$$

Note that for the task *extRating* we do not need to add the fluent $granted(a, r, extRating)$ in its preconditions as it is implicit in $invoked(a, r, creditBureau)$ because, as presented in Section 2, the task *extRating* must be executed by the same agent who performed $invoke(a, r, creditBureau)$, i.e.,

$$\textbf{nonexecutable } exec(a, r, extRating) \textbf{ if } \neg\big(p_9 \wedge \neg interrupted \wedge invoked(a, r, creditBureau)\big).$$

The execution of tasks has deterministic effects that affect the process each time the task is performed. Consider a task t, and let I (resp. O) be the set of input (resp. output) places of the related transition. The specifications of the deterministic effects of a task is $\mathcal{C}$ have the following pattern:

$$exec(a, r, t) \textbf{ causes } \bigwedge_{p\in I} \neg p \wedge \bigwedge_{p'\in O} p' \wedge executed(a, t).$$

As an example, the deterministic effects of the execution of task *intRating* are expressed in $\mathcal{C}$ with

$$exec(a, r, intRating) \textbf{ causes } p_8 \wedge \neg p_6 \wedge executed(a, intRating).$$

Some effects of a task can be nondeterministic, i.e., they can, or cannot, be true in the resulting state. As an example, the execution of task *intRating* has the nondeterministic effect of switching to true the internal rating of a customer initially

set to false. The corresponding $\mathcal{C}$ construct is

$$exec(a, r, intRating) \textbf{ may cause } intRatingPositive.$$

The execution of a task can also have conditional effects, i.e., effects which can take place or not depending on the state in which the task is executed. As an example, if the customer is industrial the task *intRating* does not only evaluate if the internal rating is positive but also establishes if the customer has a high profile, i.e., if it is convenient for the bank to reserve particular care to this customer. The result of this evaluation is nondeterministic and conditional as it only takes place if the customer is industrial. This can be expressed in $\mathcal{C}$ with

$$exec(a, r, intRating) \textbf{ may cause } highProfileIndCust \textbf{ if } isIndustrial.$$

The execution of a task can also have some indirect effects in addition to those explicitly stated, resulting from the interaction among action effects and/or static laws. As an example, the execution of task *inputCustData*, that has the nondeterministic effect of stating if the customer is industrial, has an indirect effect on the permission assignment of the RBAC model. In fact, according to the permission assignments of Table 1, the permission to execute task *intRating* is given to role *postprocessor* if the customer is industrial. This can be expressed in $\mathcal{C}$ with the law

$$\textbf{caused } pa(postprocessor, intRating) \textbf{ if } isIndustrial$$

where $pa(postprocessor, intRating)$ can be an indirect effect of the execution of *inputCustData* through its nondeterministic effect *isIndustrial*. Notice that $pa(postprocessor, intRating)$ is a *statically determined fluent*, i.e., a fluent whose value is determined by means of static laws only. Also notice that we set the value of $pa(postprocessor, intRating)$ to be false by default by means of

$$\textbf{default } \neg pa(postprocessor, intRating).$$

Laws related to an invocation of an entity have the same patterns as for the execution of a task. As an example the invocation of the Credit Bureau can be expressed in $\mathcal{C}$ by

$$\textbf{nonexecutable } invoke(a, r, creditBureau) \textbf{ if } \neg\big(p_7 \wedge highValue \wedge granted(a, r, extRating)\big) \tag{6}$$

where $invoke(a, r, creditBureau)$ is the action symbol, p_7 is the input place as shown in Fig. 1, *highValue* represents the condition associated with the arc between place p_7 and the current transition, and $granted(a, r, extRating)$ expresses the fact that the invocation must be performed by an agent granted the right to execute the task *extRating* as presented in Section 2.

Deterministic effects of the invocation of an entity can be expressed in $\mathcal{C}$ in the same way as for the invocation of tasks. As an example, the deterministic effects of the invocation of the Credit Bureau are expressed by

$$invoke(a, r, creditBureau) \textbf{ causes } p_9 \wedge \neg p_7 \wedge invoked(a, creditBureau).$$

According to (6), in our working example the invocation of the Credit Bureau cannot be executed if the preconditions do not hold; however, in case the preconditions hold, the invocation is not forced. The fact that in our working example the invocation of the Credit Bureau is triggered as soon as its preconditions hold can be expressed in $\mathcal{C}$ with

$$\textbf{nonexecutable } \neg invoke(a, r, creditBureau) \textbf{ if } p_7 \wedge highValue \wedge granted(a, r, extRating).$$

4.3. Delegation of tasks

As in Section 4.2, the preconditions for delegation express the requirements for the delegation to take place and are expressed in $\mathcal{C}$ by a formula of the form (5). The set of preconditions for the delegation of tasks through the delegation rules presented in Section 2 has a common pattern. In fact, a delegation of a task always requires that

- an agent is assigned to the role *ARole* of the rule,
- there exists a permission assignment between *Arole* and *Task*,
- an agent is assigned to the role *DRole* of the rule,
- *PreConds* are satisfied,

regardless the type of delegation. The preconditions of the rules in Table 2 in $\mathcal{C}$ have the following pattern

$$\textbf{nonexecutable } d(a_1, a_2) \textbf{ if } \neg\big(ua(a_1, Arole) \wedge ua(a_2, Drole) \wedge pa(Arole, Task) \wedge Preconds\big)$$

for all $a_1 \neq a_2$.

For example the preconditions of rule D1 are expressed with

$$\textbf{nonexecutable } d1(a_1, a_2) \textbf{ if } \neg\big(ua(a_1, supervisor) \wedge ua(a_2, postprocessor) \wedge pa(supervisor, extRating) \wedge \neg isIndustrial\big).$$

Delegation can also have implicit preconditions. In fact, the effects of a delegation may interact with different parts of the business process; as a result, a delegation can be performed not only if its preconditions are fulfilled but also if its execution does not violate existing dependencies. In $\mathcal{C}$ implicit preconditions can be expressed by means of static laws. As an example, delegation has the effect of granting the execution of a task t to an agent a by means of a role r, $granted(a, r, t)$. It appears reasonable that a delegated agent must not be disabled from executing the delegated task. In the LOP case study this can be achieved with

$$\textbf{caused } \neg granted(a, r, t) \textbf{ if } disabled(a, t) \tag{7}$$

to state that $\neg disabled(a, t)$ is an implicit precondition for the delegation of task t to agent a. Notice that this is not equivalent to add $\neg disabled(a, t)$ to the preconditions of the delegation as it corresponds to the fact that the agent is not disabled in the state before the delegation. On the other hand, (7) requires $\neg granted(a, r, t)$ whenever $disabled(a, t)$ holds.

Delegation of tasks has the deterministic effect of granting the execution of the tasks to the delegated agents. Moreover, in case of transfer delegation the agent who performed the delegation is disabled from performing the delegated task in the future. Given *Type* to be the type of delegation, $Type \in \{transfer, grant\}$, the deterministic effects can be expressed in $\mathcal{C}$ using the following pattern

$$d(a_1, a_2) \textbf{ causes } granted(a_1, Arole, Task) \wedge delegated(a_2, Arole, task) \bigwedge_{Type=trasfer} disabled(a_1, Task) \tag{8}$$

for all $a_1 \neq a_2$.

As an example, the deterministic effects of the transfer delegation rule D1 are expressed by

$$d1(a_1, a_2) \textbf{ causes } granted(a_1, supervisor, extRating) \wedge delegated(a_2, supervisor, extRating) \wedge disabled(a_1, extRating) \tag{9}$$

while the deterministic effects of the grant delegation rule D2 are expressed with

$$d2(a_1, a_2) \textbf{ causes } granted(a_1, manager, approve) \wedge delegated(a_2, manager, approve).$$

Notice that the fluent $disabled(a, extRating)$ as effect of rule D1 in (8) represents the means of transfer delegation: the agent performing the delegation is disabled from performing the task in the future.

4.4. Internal dependencies

Different elements within a business process may be characterized by dependencies. For example results of evaluations performed during the process may influence the value assumed by other fluents describing the state of the process.

In $\mathcal{C}$ such dependencies can be expressed by means of static laws. As an example, the inertial fluent *interrupted* depends on the value of other fluents as follows

$$\textbf{caused } interrupted \textbf{ if } p_9 \wedge invoked(a_1, r, creditBureau) \wedge accessed(a_2)$$

for all $a_1 \neq a_2$ and $r \neq director$, and

$$\textbf{caused } \neg interrupted \textbf{ if } p_9 \wedge invoked(a, director, creditBureau).$$

As another example, the statically determined fluent *lowRisk* has a default value false expressed with

$$\textbf{default } \neg lowRisk$$

and depends on the value of *highValue* and *intRatingPositive* through

$$\textbf{caused } lowRisk \textbf{ if } \neg highValue \wedge intRatingPositive$$

to state that the risk of a loan is low if its amount is not high and the internal rating is positive.

4.5. Workflow

As we have already seen in Section 2, the workflow of a business process manages the invocation of entities and the execution of tasks: in our work we model it by adding "dummy" activities to begin/end a flow of business activities and to skip certain operations.

In $\mathcal{C}$ these activities can be expressed by dynamic laws. The preconditions of dummy activities only deal with input places and conditions associated to inner arcs, if any. Moreover, being activities that do not require to be performed by an agent and do not perform any task, dummy activities are triggered as soon as their preconditions hold. Consider an activity e, and

let I (resp. O) be the set of input (resp. output) places of the related transition, and I' be the set of conditions on the arcs from the input places (if any). Such activities follow the pattern

$$\textbf{nonexecutable } e \textbf{ if } \neg\Big(\bigwedge_{p\in I} p \wedge \bigwedge_{f\in I'} f\Big),$$

$$\textbf{nonexecutable } \neg e \textbf{ if } \bigwedge_{p\in I} p \wedge \bigwedge_{f\in I'} f,$$

$$e \textbf{ causes } \bigwedge_{p\in I} \neg p \wedge \bigwedge_{p'\in O} p'.$$

As an example, *nop1* is expressed by

$$\textbf{nonexecutable } nop1 \textbf{ if } \neg(p_7 \wedge \neg highValue),$$

$$\textbf{nonexecutable } \neg nop1 \textbf{ if } p_7 \wedge \neg highValue,$$

$$nop1 \textbf{ causes } p_{10} \wedge \neg p_7.$$

4.6. Separation of duty constraints

SoD represents an important aspect of business processes used to prevent frauds by stating that conflicting tasks must be performed by different agents. With reference to the LOP, SoD constraints in Table 3 can be expressed in $\mathcal{C}$ by static laws constraining each set of critical tasks to be executed by at least two different agents. For example, constraint C3 is expressed by

$$\textbf{constraint } \neg\big(executed(a, r_1, prepareContract) \wedge executed(a, r_2, approve) \wedge executed(a, r_3, sign)\big).$$

4.7. Security policy

The security policy establishes which agent can perform which task. As described in Section 2, we consider a security policy based on an RBAC model enhanced with delegation and SoD constraints. We model the permission of an agent a to perform a task t with role r using the fluent $granted(a, r, t)$. Then, the RBAC model enhanced with delegation can be expressed with the law

$$\textbf{caused } granted(a, r, t) \textbf{ if } ua(a, r) \wedge activated(a, r) \wedge pa(r_1, t) \wedge senior(r, r_1) \tag{10}$$

and

$$\textbf{caused } \neg granted(a, r, t) \textbf{ if } \neg ua(a, r) \vee \neg activated(a, r) \vee \Big(\bigwedge_{r_1\in\mathcal{R}} \big(\neg pa(r_1, t) \vee \neg senior(r, r_1)\big)\Big) \textbf{ unless } delegated(a, r, t) \tag{11}$$

where $\mathcal{R}$ is the set of roles involved in the business process. Notice that (11) is made defeasible by "**unless** $delegated(a, r, t)$" to express the fact that an agent can be granted the execution of a task even if she does not fulfill the RBAC model assuming she has been delegated to execute it. Also notice that in the overall specification of a business process an agent can be granted the execution of a task only if the SoD constraints specified as presented in Section 4.6 are satisfied.

As it appears from (10), the RBAC model relies both on the user assignment, $ua(a, r)$, and the permission assignment, $pa(r, t)$. A specific, static, user assignment can be expressed in $\mathcal{C}$ by initializing the inertial fluents $ua(a, r)$ in the initial state as long as no action affects their values. In this way, the assignment of agents to roles will not change during the process execution. As an example, the static assignment of agents *davide* and *maria* given in Section 2 can be modeled as follows

$$ua(davide, director),$$

$$\neg ua(davide, r_1),$$

$$ua(maria, manager),$$

$$\neg ua(maria, r_2)$$

for all $r1 \neq director$ and $r2 \neq manager$.

A specific permission assignment can be expressed in $\mathcal{C}$ by defining the fluents $pa(r, t)$ as statically determined and specifying static causal laws for them. As an example, an excerpt of the permission assignment in Table 1 can be expressed in $\mathcal{C}$ by

default $pa(preprocessor, inputCustData)$,

default $\neg pa(postprocessor, inputCustData)$,

caused $pa(preprocessor, intRating)$ **if** $\neg isIndustrial$,

default $\neg pa(preprocessor, intRating)$,

caused $pa(postprocessor, intRating)$ **if** $isIndustrial$,

default $\neg pa(postprocessor, intRating)$.

Notice that the assignment of permissions to roles can change during the process execution according to some conditions.

The partial specifications on user and permission assignments can, in general, be modeled by means of the set of requirements they have to ensure. As an example, for the specific requirements on a user assignment defined in Section 2, the following $\mathcal{C}$ laws can be used

constraint $\neg\big(ua(a_1, director) \wedge ua(a_2, director)\big)$,

constraint $\neg\big(ua(a, director) \wedge ua(a, r)\big)$,

constraint $\neg\big(ua(a, supervisor) \wedge ua(a, postprocessor)\big)$,

constraint $\neg\big(ua(a, r_1) \wedge ua(a, r_2) \wedge ua(a, r_3)\big)$

for all $a_1 \neq a_2$, $r_1 \neq r_2$, $r_2 \neq r_3$, $r_1 \neq r_3$, and $r \neq director$.

Analogously, an excerpt of the requirements for the permission assignments defined in Section 2 can be expressed in $\mathcal{C}$ with

constraint $\neg isIndustrial \vee \neg pa(preprocessor, intRating)$,

constraint $\neg highProfileIndCust \vee \neg pa(manager, sign)$.

Notice that in this case the fluents $pa(r, t)$ are defined as exogenous.

4.8. Exceptions

A business process may also be characterized by exceptions to its normal behavior. In the presented scenario the security policy is based on an RBAC model enhanced with delegation. However, we can have exceptions to the RBAC, e.g., in case an important agent (e.g., a director) decides to do so. To model this behavior in $\mathcal{C}$ we enhance the model with a law

$$disable(a_1, a_2, t) \textbf{ causes } disabled(a_2, t) \wedge \bigwedge_{r \in \mathcal{R}} \big(\neg granted(a_2, r, t)\big) \textbf{ if } ua(a_1, director) \wedge \neg disabled(a_2, t) \tag{12}$$

where $a_1 \neq a_2$, and modifying (10) as follows

caused $granted(a, r, t)$ **if** $ua(a, r) \wedge activated(a, r) \wedge pa(r_1, t) \wedge senior(r, r_1)$ **unless** $disabled(a, t)$

to make the policy defeasible, i.e., an agent can be granted the execution of a task unless a director disables her from executing the task.

5. Experiments

Given a set of causal laws expressed in $\mathcal{C}$ and a query (i.e., a Boolean formulas built out of time-indexed facts), CCalc automatically checks whether there exist paths in the transition system specified by the action description that satisfies the query. The length of the path considered is determined by fixing the *maxstep* variable in CCalc. In particular, CCalc

1. produces a description of the transition system in the form of a set of clauses (where a clause is a disjunction of literals): assuming the head of each causal law is a literal (as in the LOP), this process is done in polynomial time through literal completion [28];
2. produces $maxstep - 1$ copies of the clauses, each copy corresponding to a time step as in planning as satisfiability [29];
3. converts each query into a corresponding set of clauses; the clauses generated so far are such that there exists a 1–1 correspondence between the paths of length *maxstep* of the transition system satisfying the query and the assignments satisfying the set of clauses;
4. calls a SAT solver to determine an assignment satisfying the set of clauses (if any);
5. if the SAT solver returns a satisfying assignment, then the corresponding path is returned to the user.

Notice that there is no 1–1 correspondence between the paths in the extended elementary net of the LOP and the paths in the transition diagram corresponding to the formalization of the LOP as an action description. Indeed, in our formalization we have transitions from one state to the state itself because no action is performed. These loops, called *stuttering steps* (cf., p. 17 in [30]), can be practically useful because they allow us to have paths in the transition diagram with length *maxstep* corresponding to paths in the extended elementary net with length $\leqslant$ *maxstep*. (Because of the security policy, the loop in the LOP can be executed at most once, and thus $maxstep = 14$.) Considering that only p_1 is true in the initial states of the transition diagram, we take into account all the possible paths in the extended elementary net leaving from p_1. Other conditions on paths are imposed depending on the specific problem we consider, detailed in the following subsections. The complete specification is available at http://www.ai-lab.it/serena/jcss.txt.

In the rest of this section we describe how we have used CCalc (i) to establish whether the control flow together with the security policy meets the expected security properties, (ii) to synthesize a security policy for the business process under given security requirements, and (iii) to find a resource allocation plan ensuring the process executability according to the given security policy.

5.1. Verification of security properties

The security policy of a business process manages the access of agents to tasks, and should ensure that undesirable behaviors, e.g., frauds, do not occur. In this paper the security policy is given in terms of an RBAC model enriched with delegation rules and object-based SoD constraints. Because of the complexity of the resulting specifications, and the interplay with the security policy, it may not be trivial to establish if other desirable security properties hold (e.g., because entailed by the already enforced security policy) or if it is necessary to revise the model and/or the security policy in order.

In our first experiment we fed CCalc with the specification of the LOP given in Section 4 featuring a security policy characterized by

- a specific RBAC model with the permission assignment of Table 1 and the user assignment presented in Section 2;
- the delegation rules in Table 2; and
- the SoD constraints in Table 3,

and we considered the following property:

> "If the process terminates successfully, then no single agent has performed all the tasks *intRating*, *extRating* if *highValue*, *approve*, and *sign*."

Thus, we used CCalc to determine whether there exists a path leading to a state in which the same agent performs the tasks *intRating*, *extRating* if *highValue*, *approve*, and *sign*, i.e., where the following property is satisfied:

$$p_{18} \wedge productApproved \wedge executed(a, intRating) \wedge \big(\neg highValue \vee executed(a, extRating)\big) \wedge executed(a, approve) \wedge executed(a, sign) \tag{13}$$

CCalc found the trace, an excerpt of it is reported in Fig. 3. The trace shows that (13) is satisfied, i.e., the violation occurs, if the loan amount is not high, i.e., $\neg$*highValue*, and thus the external rating is not evaluated. The perpetrator for the violation is *stefano*, who executes *intRating* as *supervisor* and can nevertheless execute *approve* and *sign* by means of delegation. In fact a manager, *marco*, delegates him to approve the document by means of the delegation rule D2 and the director delegates him to sign the contract by means of the delegation rule D3. By inspecting the intermediate states of the trace it is easy to conclude that the violation occurs if the internal rating is positive and the customer is industrial with a high profile.

To avoid this violation we restricted the applicability conditions of delegation rule D2 by conjoining it with the fact *highValue*. In fact, when the loan has not a high value, the security policy is less restrictive and the application of D2 must be prevented. CCalc does not find any violation in the specification modified in this way.

The verification of SoD properties over a version of the LOP is carried out by using NuSMV in [5]. However the case study considered is rather different, e.g., the workflow (which involves different tasks) structure and the delegation model are different and no exceptions to the RBAC security policy with delegation are considered. Moreover, the specification of the case study and the experimental analysis performed in [5] are not mature, e.g. though an RBAC policy is considered, the assignment between agents and tasks, via roles, is static. Thus this work can be compared with ours w.r.t. the overall approach and the expressiveness of the language used but it is not possible a comparison of the analysis results.

5.2. Synthesis of the permission assignment

In our second experiment we synthesize the permission assignment for an RBAC model for the LOP given the requirements for the user and the permission assignment presented in Section 2. Note that the approach we use is general as it relies on the idea of partially defining an aspect of the business process, i.e., only defining a set of requirements we want

```
                              ...
4: granted(stefano,supervisor,intRating) isIndustrial p5  p6
   p7 ...

ACTIONS: exec(stefano,supervisor,intRating)  nop1 ...

5:  isIndustrial  intRatingPositive  highProfileIndCust lowrisk
    executed(stefano,intRating)  p5  p8  p10 ...
                              ...
7: isIndustrial  intRatingPositive  highProfileIndCust
   ua(marco,manager) ua(stefano,supervisor) lowrisk
   executed(stefano,intRating) p12 pa(manager,approve) ...

ACTIONS:  d2(marco,stefano) ...

8: granted(stefano,manager,approve) executed(stefano,intRating)
   delegated(stefano,manager,approve)  p12 lowrisk  ...

ACTIONS: exec(stefano,manager,approve)  ...

9: isIndustrial  intRatingPositive  highProfileIndCust
   productApproved executed(stefano,intRating) lowrisk
   executed(stefano,approve)  p13 ...
                              ...
10: isIndustrial  intRatingPositive  highProfileIndCust lowrisk
    productApproved ua(davide,director) ua(stefano,supervisor)
    executed(stefano,intRating) executed(stefano,approve)
    p14  p15  pa(director,sign)  ...

ACTIONS: d3(davide,stefano) ...

11: granted(stefano,director,sign) isIndustrial lowrisk
    intRatingPositive highProfileIndCust  productApproved
    executed(stefano,intRating) executed(stefano,approve)
    delegated(stefano,director,sign) p14  p15 ...

ACTIONS: exec(stefano,director,sign) ...

12: isIndustrial  intRatingPositive  highProfileIndCust lowrisk
    executed(stefano,intRating) executed(stefano,approve)
    executed(stefano,sign) productApproved p15  p16   ...
                              ...
14: isIndustrial  intRatingPositive  highProfileIndCust
    productApproved executed(stefano,intRating) lowrisk
    executed(stefano,approve)  executed(stefano,sign) p18   ...
```

Fig. 3. An excerpt of the trace violating property (13) found by CCALC.

it to satisfy, and automatically synthesizing a specific instance that ensures the process executability and satisfies the requirements. As an example, in this experiment we consider the problem of synthesizing the permission assignment but the approach could be used as well to synthesize the user assignment.

We fed CCALC with the specification of the LOP given in Section 4 featuring a security policy characterized by

- a generic RBAC model with the requirements for the user and permission assignments given in Section 2; and
- the SoD constraints in Table 3,

and we considered the problem of finding a permission assignment that ensures a successful process execution. The process ends successfully if it reaches a state in which both p_{18} and *productApproved* hold regardless of the value of *intRatingPositive*, *isIndustrial*, *highValue*, and *highProfileIndCust* and the fact that the process has been interrupted. We have therefore run CCALC against all possible scenarios requiring $p_{18} \wedge$ *productApproved* and we extracted the permission assignment from the execution traces returned by the tool. Additionally, we looked for a permission assignment such that the process can be executed involving the minimal number of agents.

SoD constraints require at least two agents for performing the critical tasks. CCALC reports that it is not possible to find an execution path for all the scenarios with two agents. Thus, we considered three agents. In this case CCALC was able to find an execution trace for all the scenarios. We then extracted from the execution traces returned by the tool the

Table 7
Permission assignment for the LOP automatically synthesized.

Task	Role
inputCustData	*preprocessor*
prepareContract	*preprocessor*
intRating	if (*isIndustrial*) then *postprocessor* else *preprocessor*
extRating	if (*interrupted*) then *director* else *preprocessor*
approve	*director*
sign	*director*
createAccount	*postprocessor*

Table 8
Resource allocation plan for all scenarios where the process is interrupted.

Agent	Roles	Tasks
a_1	*preprocessor*, *supervisor*	*inputCustData*, *intRating*, *prepareContract*, *createAccount*
a_2	*director*	*extRating*, *approve*, *sign*

Table 9
Resource allocation plan for all scenarios where ¬*highValue*, *intRatingPositive*, *isIndustrial*, and ¬*highProfileIndCust*.

Agent	Roles	Tasks
a_1	*manager*, *supervisor*	*intRating*, *approve*, *sign*
a_2	*preprocessor*, *supervisor*	*inputCustData*, *prepareContract*, *createAccount*

Table 10
Resource allocation plan for all scenarios where ¬*highValue*, *intRatingPositive*, ¬*isIndustrial*, and ¬*highProfileIndCust*.

Agent	Roles	Tasks
a_1	*manager*, *postprocessor*	*prepareContract*, *createAccount*
a_2	*preprocessor*, *manager*	*inputCustData*, *intRating*, *approve*, *sign*

permission assignment through the values of the fluents $pa(r, t)$. The permission assignment we extracted is reported in Table 7.

5.3. Resource allocation plan

Our last experiment was to determine a resource allocation plan ensuring a successful process completion according to the security policy.

We fed CCALC with the specification of the LOP given in Section 4 featuring a security policy characterized by

- the specific permission assignment in Table 1;
- a generic user assignment with the requirements given in Section 2;
- the delegation rules in Table 2 with the applicability conditions of D2 restricted according to Section 5.1; and
- the SoD constraints in Table 3,

and we considered the problem of finding for all possible scenarios a resource allocation plan that ensures a successful process completion, i.e., $p_{18} \wedge productApproved$. Notice that as a specific assignment of agents to roles is not given, a possible agent-role assignment can also be obtained from the execution trace returned by CCALC.

As in Section 5.2, the process can take place in different scenarios characterized by different nondeterministic effects. As a result, we run CCALC against all possible successful process completions and we extracted a resource allocation plan (i.e., an assignment of agents to tasks) from the execution trace returned by the tool. We considered the further requirement of finding resource allocation plans involving the minimal number of agents.

Considering two agents, say a_1 and a_2, CCALC found an execution trace for all the scenarios where

- the process is interrupted; or
- *intRatingPositive* and *isIndustrial* hold while *highValue* and *highProfileIndCust* do not; or
- *intRatingPositive* holds while *highValue*, *isIndustrial*, and *highProfileIndCust* do not.

The resource allocations found by CCALC for each of these scenarios are reported in Tables 8, 9, and 10, respectively. However CCALC could not find a unique execution trace for all other scenarios involving only two agents.

Table 11
Resource allocation plan for all scenarios where *highValue* and ¬*highProfileIndCust*.

Agent	Roles	Tasks
a_1	*preprocessor*, *supervisor*	*inputCustData*, *intRating*, *prepareContract*
a_2	*manager*, *supervisor*	*extRating*, *createAccount*, *sign*
a_3	*director*	*approve*

Table 12
Resource allocation plan for all scenarios where *highValue* and *highProfileIndCust*.

Agent	Roles	Tasks
a_1	*preprocessor*, *postprocessor*	*inputCustData*, *intRating*, *createAccount*
a_2	*preprocessor*, *supervisor*	*extRating*, *prepareContract*
a_3	*director*	*approve*, *sign*

Table 13
Resource allocation plan for all scenarios where ¬*highValue* and *highProfileIndCust*.

Agent	Roles	Tasks
a_1	*preprocessor*, *supervisor*	*inputCustData*, *sign*
a_2	*manager*, *supervisor*	*intRating*, *prepareContract*, *createAccount*
a_3	*director*	*approve*

Table 14
Resource allocation plan for all scenarios where ¬*highValue*, ¬*intRatingPositive*, and ¬*highProfileIndCust*.

Agent	Roles	Tasks
a_1	*preprocessor*, *postprocessor*	*inputCustData*, *createAccount*
a_2	*preprocessor*, *postprocessor*	*intRating*, *prepareContract*
a_3	*director*	*approve*, *sign*

Thus, we considered three agents and CCalc found an execution trace for all the remaining scenarios. The resource allocation plans extracted from the execution traces are reported in Tables 11, 12, 13, 14. In this way we obtained assignments of agents to roles for every scenario in which the process can be executed. Furthermore, we found that the minimal number of agents depends on the specific scenario and, for each scenario, we also obtained the assignments of agents to roles that allow them to complete the process.

As far as computational aspects of our approach are concerned, we herewith briefly mention what are the CPU times that CCalc spent for solving the three tasks of interest. On a Pentium IV 1.6 GHz machine with 3GB of RAM, CCalc takes from 30 to 50 seconds to ground the specifications in input and create the corresponding set of clauses. A satisfying assignment is then found in negligible time by running a modern, e.g., MINISAT, SAT solver on ϕ.

6. A comparison between $\mathcal{C}$ and the SMV language

In this section we analyze the languages $\mathcal{C}$ and SMV, which are supported by CCalc and NuSMV respectively. The comparison focuses on the ability of the two languages to manage changes and updates of the specifications.

Here we provide a very brief introduction of the SMV language focusing on the aspects that are relevant for our comparison. A detail account of the language can be found in [31]. In an SMV specification the system under design is represented as a Kripke structure defined by state variables and transitions. State variables are defined using the keyword **VAR** and, for the scope of our analysis, are boolean. We also consider input variables (defined using the keyword **IVAR**) that are used to represent values given as input to the system and are also used to label transitions. SMV supports two styles for declaring transitions: the assignment style and the constraint style. The *assignment style* is based on assignments of initial and next values of each variable using the keyword **ASSIGN**. (If no assignment is specified for a variable, then the value of the variable evolves nondeterministically.) The *constraint style* allows to specify conditions over variable values on the initial states, the states, and the transitions by using the constructs **INIT**, **INVAR** and **TRANS**, respectively. In our comparison we use the constraint style as it is more flexible. Moreover, specifications in assignment style can be easily rewritten as an equivalent specification in constraint style.

$\mathcal{C}$ and SMV differ in fundamental way. In $\mathcal{C}$, if there is no cause for a fact, the fact can be neither true nor false (and thus the formula corresponding to the specification is unsatisfiable). In SMV declared facts are exogenous. The different semantics given to the facts declared has a considerable impact on the way a model can be specified to obtain the same behavior in both languages. As an example consider a simple scenario (1) where an agent is granted the execution of a task

```
1  simpleFluent potentialOwner, delegated, unassigned, granted,
2  exogenous potentialOwner, delegated, unassigned,
3  caused granted if potentialOwner,
4  caused granted if delegated,
5  caused ¬granted if unassigned.
```

Fig. 4. $\mathcal{C}$ specification of scenario (1).

```
1  VAR
2  potentialOwner : boolean,
3  delegated : boolean,
4  unassigned : boolean,
5  granted : boolean,
6  INVAR potentialOwner → granted,
7  INVAR delegated → granted,
8  INVAR unassigned →!granted,
9  INVAR potentialOwner|delegated|unassigned.
```

Fig. 5. SMV specification of scenario (1).

```
1  simpleFluent potentialOwner, delegated, unassigned, granted,
2  exogenous potentialOwner, delegated, unassigned,
3  exogenousAction disable,
4  caused granted if potentialOwner,
5  caused granted if delegated,
6  caused ¬granted if unassigned,
7  disable causes ¬granted.
```

Fig. 6. $\mathcal{C}$ specification of scenario (2).

```
1  VAR
2  potentialOwner : boolean,
3  delegated : boolean,
4  unassigned : boolean,
5  granted : boolean,
6  IVAR
7  disable : boolean,
8  INVAR potentialOwner → granted,
9  INVAR delegated → granted,
10 INVAR unassigned →!granted,
11 TRANS disable → next(granted) = 0,
12 INVAR !(potentialOwner|delegated|unassigned) → disable.
```

Fig. 7. SMV specification of scenario (2).

if she is a potential owner of the task (*potentialOwner*) or delegated to perform it (*delegated*), while she is not granted the execution if she is not authorized to execute it (*unassigned*).

This scenario can be described in $\mathcal{C}$ by the specification in Fig. 4 where either *potentialOwner*, or *delegated*, or *unassigned* have to hold in order to provide a cause for *granted* to be either true or false. To specify the same behavior in SMV it is necessary to explicitly state the dependencies that exist between the facts *potentialOwner*, *delegated*, and *unassigned* as they all determine the value of *granted* as shown in Fig. 5. A straightforward consequence is that while modifications to the $\mathcal{C}$ specification of Fig. 4 can be done incrementally, this is not the case for the SMV specification. As an example we can imagine to extend the previous scenario by introducing a mechanism that according to some conditions (abstracted away in our example) disables an agent, i.e. prevents the execution of tasks, namely a scenario (2). In $\mathcal{C}$ this can be specified as shown in Fig. 6 by adding to the specification in Fig. 4 an action *disable* that causes *granted* to be false, i.e. incrementally adding the lines 3 and 7 in Fig. 6. In SMV this new scenario can be expressed as shown in Fig. 7. It is clear that the SMV specification is not incremental and the fact that the action *disable* is executed if neither *potentialOwner*, *delegated*, nor *unassigned* hold must be explicit (line 12 of Fig. 7).

Many behaviors that can be obtained implicitly in $\mathcal{C}$ specifications must be made explicit in SMV. As a further example consider a scenario (3) where agents are not granted the execution of tasks by default unless they are assigned this duty by, e.g., an administrator, or they are delegated. This scenario can be specified in $\mathcal{C}$ as shown in Fig. 8. In this example, *assigned* is exogenous while *granted* has a negative default value unless *assigned* or *delegate* cause it to be true. This behavior can be specified in SMV as in Fig. 9 by explicitly stating when the value of *granted* is negative (line 8 in Fig. 9). Imagine that we now want to modify the scenario by allowing an agent granted the execution of a task to retain this authorization, namely a scenario (4). In $\mathcal{C}$ this can be simply done by incrementally adding **inertial** *granted* to the specification in Fig. 8. The main difference of the new specification is that after *granted* has become true, either because *delegate* has been executed or *assigned* was true, it can retain the true value other than become false because of the default. To apply the same modification in SMV the specification changes considerably in the way *granted* is specified (lines 8 and 9 in Fig. 10).

```
1 simpleFluent granted, assigned,
2 exogenous assigned,
3 default ¬granted,
4 exogenousAction delegate,
5 caused granted if assigned,
6 delegate causes granted.
```

Fig. 8. SMV specification of scenario (3).

```
1 VAR
2 granted : boolean,
3 assigned : boolean,
4 IVAR
5 delegate : boolean,
6 INVAR assigned → granted,
7 TRANS delegate → next(granted) = 1,
8 TRANS !assigned & !delegate → !granted.
```

Fig. 9. $\mathcal{C}$ specification of scenario (3).

```
1 VAR
2 granted : boolean,
3 assigned : boolean,
4 IVAR
5 delegate : boolean,
6 INVAR assigned → granted,
7 TRANS delegate → next(granted) = 1,
8 TRANS granted → next(granted) = 1|next(granted) = 0,
9 TRANS !granted & !assigned & !delegate →!granted.
```

Fig. 10. SMV specification of scenario (4).

Similar scenarios occur also in the LOP case study presented in Section 2. As an example we can consider the specification of the authorization to execute task *extRating* with respect to delegation. In this case it is again possible to observe that SMV requires to explicitly state all the mechanism which can be left implicit in $\mathcal{C}$. In particular, in the $\mathcal{C}$ specification presented in Section 4, $delegated(a, r, t)$ prevents the authorization to execute to be denied when an agent is delegated. More in detail, $delegated(a, r, t)$ is an inertial fact which is set to true after D1 is executed and has a false default value through the **unless** abbreviation in (11), and thus its behavior resembles the one of specification in Fig. 8 after the modification according to the scenario (4). As a consequence, its translation in SMV recalls the specification in Fig. 10. Notice that SMV is a propositional language and thus each $\mathcal{C}$ predicate has to be specified for each ground instance, e.g.

$$
\begin{aligned}
&\textbf{INIT} \quad !delegated(pierSilvio, pierPaolo, supervisor, extRating);\\
&\textbf{TRANS} \quad d1\_pierSilvio\_pierPaolo\_supervisor\_extRating \rightarrow\\
&\qquad \textbf{next}(delegated\_pierSilvio\_pierPaolo\_supervisor\_extRating) = 1;\\
&\textbf{TRANS} \quad delegated\_pierSilvio\_pierPaolo\_supervisor\_extRating \rightarrow\\
&\qquad \textbf{next}(delegated\_pierSilvio\_pierPaolo\_supervisor\_extRating) = 1 |\\
&\qquad \textbf{next}(delegated\_pierSilvio\_pierPaolo\_supervisor\_extRating) = 0;\\
&\textbf{TRANS} \quad !delegated\_pierSilvio\_pierPaolo\_supervisor\_extRating \;\&\\
&\qquad !d1\_pierSilvio\_pierPaolo\_supervisor\_extRating \rightarrow\\
&\qquad \textbf{next}(delegated\_pierSilvio\_pierPaolo\_supervisor\_extRating) = 0;
\end{aligned}
$$

where the ground predicates, e.g. $delegated(pierSilvio, pierPaolo, supervisor, extRating)$, stand for the corresponding propositional letters.

7. Related work

A preliminary version of this work is [1]. Here we have significantly extended [1] by providing: (i) a mapping from elementary net and security policy into $\mathcal{C}$, which also clearly shows how the specification of the workflow and the security policy can be kept separate; (ii) an extended experimental analysis; (iii) a comparison between $\mathcal{C}$ and SMV, which highlights some of the main features of the language $\mathcal{C}$; (iv) an exhaustive analysis of the related works that covers all main aspects of our work; and (v) a detailed description of the reasoning tasks we analyze.

7.1. Action-based specification languages and reasoning tools

Action languages [11] serve for describing changes that are caused by performing actions. As predecessor of $\mathcal{C}$, action language $\mathcal{A}$, i.e., the propositional fragment of ADL, was presented in [32] to enhance the expressive power of the STRIPS planning language [33] by allowing conditional effects. Action language $\mathcal{B}$ extended $\mathcal{A}$ with static laws. In action language $\mathcal{C}$ nondeterministic actions and the concurrent execution of actions are more conveniently described than in $\mathcal{B}$.[4] Moreover, with $\mathcal{C}$ we are free to decide for each fluent whether or not to postulate inertia for it. Given it is based on the theory of causal explanation proposed in [35], $\mathcal{C}$ distinguishes between asserting that a fact "simply" holds, and the stronger assertion that "it is caused". $\mathcal{C}+$ [28] further extended $\mathcal{C}$ by allowing fluents to be multi-valued, other than some other (minor) changes. The language $\mathcal{C}+$ has been then extended in several directions, e.g., to include "additive fluents" [36], to represent numeric-valued fluents [37] and to include the possibility of referring to other action descriptions in the definition of a new action domain [38]. The applicability of action language $\mathcal{C}/\mathcal{C}+$ spans from the representation of "classical" AI problems [39], to planning [40], multi-agent domains [41] and robotics [42], coupled with the reasoning capabilities of CCalc. $\mathcal{C}$ has been also used as "basic language" to compare [43] and update [44] action domain descriptions. As far as other knowledge representation languages are concerned, we herewith mention three more languages. The language $\mathcal{K}$ [10] is a declarative planning language based on principles and methods of logic programming. Its distinguished feature is that $\mathcal{K}$ describes transitions between "states of knowledge" rather than between "states of the world" used in language $\mathcal{C}/\mathcal{C}+$. Like $\mathcal{C}/\mathcal{C}+$, it uses a notion of causation. Correspondences between (fragments of) $\mathcal{K}$ and $\mathcal{C}$ are presented in the mentioned paper. Temporal Action Logics Language *TAL* [12] uses a surface language $\mathcal{L}(ND)$ (Narrative Description Language) to provide a high-level notation of "narratives", i.e., collections of statements. There is the possibility to describe both the static and dynamic aspects of a narrative. A narrative described in $\mathcal{L}(ND)$ is translated, after some steps, into a logically equivalent first order theory. Dedicated algorithms are then used to reason about narratives. Finally, in [45] Concurrent Transaction Logic ($\mathcal{CTR}$) is proposed, and used as a language for specifying, analyzing and scheduling of workflows.

Action language $\mathcal{C}$ and system CCalc have been already used in the context of business processes [46,47]. In [46] the research objective is to describe, simulate, compose, verify, and develop Web services. However, this work does not consider (complex) security policies to be modeled or verified. On the contrary, our approach takes into account security policies and focuses on modeling and reasoning about business processes with the objectives of verifying security properties, synthetizing policies, and finding resource allocation plans. In [47] the author considers activities with duration and the cost of a workflow execution but, differently from our approach, he does not take into account complex security policies (i.e., agents are statically assigned to tasks) and mandatory requirements, e.g., SoD. This means that it does not consider delegations either. The model is simpler than the one we consider, e.g., indirect effects are not considered.

7.2. Automatic verification of business processes under authorization constraints

The use of model checking for the automatic analysis of business processes has been put forward and investigated in [5]. The paper shows that business processes with RBAC policies and delegation can be formally specified as transition systems and that SoD properties can be formally expressed as Linear Temporal Logic formulas specifying the allowed behaviors of the transition systems. The viability of the approach is shown through its application to a version of the LOP and the NuSMV [24] model checker is used to carry out the verification. However the LOP version considered in [5] is much simpler and does not feature many of the aspects we consider, as already underlined in Section 5.1. Our approach, by using an action language, allows for a more natural and concise modeling of the business process and of the associated security policy. Moreover, our approach allows for the separate specification of the workflow and of the security policy while this is not the case for the approach presented in [5], where even small changes in the workflow or in the security policy may affect the specification of the whole transition system. Our approach therefore considerably simplifies the specification process, thereby reducing the probability of introducing bugs in the specification.

A formal framework that uses the SAL model checker [48] to analyze SoD properties as well as to synthesize a resource allocation plan for a business process has been presented in [6]. However, differently from our approach, this framework does not offer a natural modeling and RBAC is the only access control model supported (with tasks rigidly associated with specific roles). Moreover, this work does not take into account the global state of the process and assumes an interleaving semantics. This is not the case in our approach as it accounts for a global state that can be affected by the execution of the tasks as well as for multiple actions to be executed simultaneously.

An approach to the combined modeling of business workflows with RBAC models is presented in [7]. The paper proposes an extended finite state machine model that allows for the model checking of SoD properties by using the model checker SPIN [49]. It considers a simple RBAC model only based on previous activation (or non-activation) of roles and it does not take into account delegation. A number of techniques to alleviate the state explosion problem are presented.

Some computational techniques for analyzing SoD by integrating workflows of the enterprise processes into the RBAC framework is presented in [21]. It proposes an algorithm for generating mutually exclusive roles (MER) to enforce SoD and verification algorithms to check if a role authorization together with a user-role assignment satisfy static and dynamic SoD

[4] In [34] it is presented an alternative extension of the language $\mathcal{A}$ to enable concurrent execution of actions.

constraints. The algorithm proposed to check for dynamic SoD also returns a set of user-role activations that guarantees to satisfy the SoD constraints. However the approach does not allow for the separate specification of the workflow and the security policy as the workflow in input to the algorithms captures aspects related to the policy, e.g. a conditional permission assignment has to be captured by duplicating the task and adding an if condition to the workflow structure. Moreover it does not consider hierarchy of roles that by allowing senior roles to perform tasks assigned to junior roles would considerably affect the proposed approach. Also notice that MER is a way to enforce a subset of SoD that, e.g., is not sufficient for the kind of SoD constraints we consider in our work. Finally, [21] does not consider the problem of synthesizing the permission assignment relation while this is the case in our approach.

The paper [22] proposes to model workflows and security policies in a notation based on Colored Petri nets, with an automatic translation from the process model into a specification language and the usage of SPIN to verify SoD properties. However, no provision is made for the assignment of an agent to multiple roles, role hierarchy, delegation, and the global state of the process.

Other approaches that use (Colored) Petri nets in this context are [50,51]. The first paper introduced a model for SoD in workflows that are specified with Petri nets, and allows for simulating and analyzing workflows and security rules at build time; rules are given as facts of a logic program and expressed in propositional logic. However it does not take into account, e.g., role hierarchy or conditions in the permission assignment and the reasoning task of synthesizing the permission assignment. [51], which is much more recent than [50], presents a formal technique to model and analyze RBAC using Colored Petri nets (CP-nets). The resulting CP-net model can be composed with context-specific aspects of the application of interest, e.g. the workflow, however the approach focuses on the access control policy and does not allow for the separate specification of the workflow and the access control policy in a modular way. Moreover the use of CP-nets does not allow the use of advanced features as, e.g., implicit preconditions of tasks. A graphical representation of the CP-nets models can be provided by using CPN tools.[5] However the approach does consider the problems of synthesizing permission assignments and resource allocation plans.

A security validation approach for business processes that employs state-of-the-art model checking techniques and makes them usable by business analysts is presented in [52]. The paper considers business processes expressed in BPMN enhanced with an application-dependent security policy, e.g. delegation of tasks, agent substitution, and fully automates their translation into a formal model suitable to formal analysis while offering graphical user interfaces to define security properties and easy-to-understand feedback for the business analysts. However the approach does not consider some business processes features which are relevant for an accurate security analysis, e.g., nondeterministic, indirect and conditional effects of tasks. This is due to the fact that these aspects are not directly available in the industrially suited languages, as BPMN, as their focus is on the definition of the procedural behavior of the process, e.g. they define as output of a task all the data that can be affected but they do not specify how and when their values can be affected. It would thus be of great interest to (i) evaluate extensions to the languages used in the industrial environments in order to support the declarative specification of advanced, relevant features and (ii) automate the process of transformation of the extended languages into $\mathcal{C}$. As an example, BPMN could be enhanced with annotations supporting a declarative, $\mathcal{C}$-like specification of, e.g., nondeterministic, indirect and conditional effects of tasks. The resulting model could then be automatically translated into a $\mathcal{C}$ specification as the one presented in Section 4. Notice that the automation of the translation would make it language- and application-dependent, e.g. [52] relies on BPMN and the industrial environment SAP Netweaver BPM [53]. We leave these research directions for future work.

7.3. Frameworks for security policies

An approach based on model checking for the analysis and synthesis of complex security policies is presented in [19] and further developed in [20]. In particular, they consider fine-grained policies where permissions are the ability of agents to access resources to read or write. They propose the RW (Read and Write) access control formalism based on propositional logic and define a machine-readable language to express policies modelled in the RW formalism and properties to be verified against the model. The RW model checking algorithm and a related tool which implements the algorithm are presented. The proposed framework is expressive and allows for the specification of administrative policies either. However, there is no way to express mutual exclusions between the values of different variables, e.g., it is not possible to express a rule of the form "an agent cannot be a student and a lecturer at the same time", and there is no way to express inheritance between roles. Moreover, such works do not take into account the natural interactions between the workflow and the security policy while this is the case in our approach.

In [54–57] rule-based policy specifications are presented and evaluated. In particular, [57] first presents a broad view of different policy languages, and then a system for specifying and enforcing security and private policy, called PROTUNE [56]. However the focus of these works is on policies languages and enforcement from a run-time point of view, usually in the context of the Semantic Web, i.e., they evaluate the existing languages and propose a system for specifying and cooperatively enforcing security and privacy policies in open environments such as the Web, where parties may get in touch without being previously known to each other. On the contrary our work focuses on security policies in the context of business processes

[5] See, e.g., http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

and thus w.r.t. the execution of tasks within a workflow and on a security analysis to be performed prior to the deployment of the system.

8. Conclusions

The design and verification of business processes under authorization constraints is a time consuming and error-prone activity. Moreover, due to the complexity that business processes subject to a security policy may reach, it can be difficult to verify even basic properties such as the executability of the process w.r.t. the available resources by manual inspection only, or by simulation.

In this paper we have presented an action-based approach to the formal specification and automatic analysis of business processes under authorization constraints. With our approach we have been able to both greatly simplify the specification activity, and allow for the separate specification of the workflow and of the associated security policy, while retaining the ability to perform a fully automatic analysis of the specifications by using the Causal Calculator CCalc. The experiments we have presented indicate that our approach can be profitably used to execute a number of reasoning tasks particularly important from the application viewpoint, such as verify the process executability, synthesize a permission assignment and identify resource allocation plans complying with the security policy.

Acknowledgments

This work is partially supported by the FP7-ICT-2007-1 Project No. 216471, "AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures" (*www.avantssar.eu*) and by the project "Automated Security Analysis of Identity and Access Management Systems (SIAM)" funded by Provincia Autonoma di Trento in the context of the "Team 2009—Incoming" COFUND action of the European Commission (FP7).

References

[1] A. Armando, E. Giunchiglia, S.E. Ponta, Formal specification and automatic analysis of business processes under authorization constraints: An action-based approach, in: S. Fischer-Hübner, C. Lambrinoudakis, G. Pernul (Eds.), Proc. of 6th International Conference on Trust, Privacy & Security in Digital Business (TrustBus'09), in: Lecture Notes in Comput. Sci., vol. 5695, Springer, 2009, pp. 63–72.
[2] J.L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice Hall PTR, 1981.
[3] OMG, Business Process Modeling Notation (BPMN), http://www.omg.org/spec/BPMN/2.0, 2009.
[4] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, C.E. Youman, Role-based access control models, Computer 29 (2) (1996) 38–47.
[5] A. Schaad, V. Lotz, K. Sohr, A model-checking approach to analysing organisational controls in a loan origination process, in: D.F. Ferraiolo, I. Ray (Eds.), Proc. of 11th ACM Symposium on Access Control Models and T (SACMAT 2006), ACM, 2006, pp. 139–149.
[6] A. Cerone, Z. Xiangpeng, P. Krishnan, Modelling and resource allocation planning of BPEL workflows under security constraints, Tech. Rep. 336, UNU-IIST, http://www.iist.unu.edu/, June 2006.
[7] A. Dury, S. Boroday, A. Petrenko, V. Lotz, Formal verification of business workflows and role based access control systems, in: L. Peñalver, O.A. Dini, J. Mulholland, O. Nieto-Taladriz (Eds.), Proc. of the First International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2007), 2007, pp. 201–210.
[8] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2003.
[9] M. Gelfond, V. Lifschitz, Representing action and change by logic programs, J. Log. Program. 17 (2/3&4) (1993) 301–321.
[10] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres, A logic programming approach to knowledge-state planning: Semantics and complexity, ACM Trans. Comput. Log. 5 (2) (2004) 206–263.
[11] M. Gelfond, V. Lifschitz, Action languages, Electron. Trans. Artif. Intell. 2 (1998) 193–210.
[12] P. Doherty, J. Gustafsson, L. Karlsson, J. Kvarnström, Tal: Temporal action logics language specification and tutorial, Electron. Trans. Artif. Intell. 2 (1998) 273–306.
[13] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres, A logic programming approach to knowledge-state planning: Semantics and complexity, ACM Trans. Comput. Log. 5 (2) (2004) 206–263.
[14] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artificial Intelligence 138 (2002) 181–234.
[15] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Trans. Comput. Log. 7 (3) (2006) 499–562.
[16] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres, A logic programming approach to knowledge-state planning, II: The dlv^k system, Artificial Intelligence 144 (1–2) (2003) 157–211.
[17] Texas Action Group at Austin, The causal calculator, available at http://www.cs.utexas.edu/users/tag/cc/, 2009.
[18] E. Giunchiglia, V. Lifschitz, An action language based on causal explanation: Preliminary report, in: Proc. of the Fifteenth National Conference on Artificial Intelligence (AAAI 1998), AAAI Press, 1998, pp. 623–630.
[19] D.P. Guelev, M. Ryan, P.-Y. Schobbens, Model-checking access control policies, in: K. Zhang, Y. Zheng (Eds.), Proc. of 7th International Conference on Information Security (ISC 2004), in: Lecture Notes in Comput. Sci., vol. 3225, Springer, 2004, pp. 219–230.
[20] N. Zhang, M. Ryan, D.P. Guelev, Synthesising verified access control systems through model checking, J. Comput. Secur. 16 (1) (2008) 1–61.
[21] R. Hewett, P. Kijsanayothin, A. Thipse, Security analysis of role-based separation of duty with workflows, in: Proc. of the Third International Conference on Availability, Reliability and Security (ARES 2008), IEEE Computer Society, 2008, pp. 765–770.
[22] C. Wolter, P. Miseldine, C. Meinel, Verification of business process entailment constraints using SPIN, in: F. Massacci, S.T. Redwine Jr., N. Zannone (Eds.), Proc. of the First International Symposium on Engineering Secure Software and Systems (ESSoS 2009), in: Lecture Notes in Comput. Sci., vol. 5429, Springer, 2009, pp. 1–15.
[23] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NuSMV: A New Symbolic Model Verifier, Lecture Notes in Comput. Sci., vol. 1633, 1999, http://nusmv.irst.itc.it/index.html.
[24] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: E. Brinksma, K.G. Larsen (Eds.), 14th International Conference on Computer Aided Verification (CAV 2002), in: Lecture Notes in Comput. Sci., vol. 2404, Springer, 2002, pp. 359–364.

[25] S. Frau, R. Gorrieri, C. Ferigato, Petri net security checker: Structural non-interference at work, in: P. Degano, J.D. Guttman, F. Martinelli (Eds.), Formal Aspects in Security and Trust, in: Lecture Notes in Comput. Sci., vol. 5491, Springer, 2008, pp. 210–225.
[26] V. Atluri, J. Warner, Supporting conditional delegation in secure workflow management systems, in: E. Ferrari, G.-J. Ahn (Eds.), Proc. of 10th ACM Symposium on Access Control Models and Technologies (SACMAT 2005), ACM, 2005, pp. 49–58.
[27] J. Crampton, H. Khambhammettu, Delegation in role-based access control, in: Proc. of 13th European Symposium on Research in Computer Security (ESORICS 2006), in: Lecture Notes in Comput. Sci., vol. 5283, Springer, 2006, pp. 174–191.
[28] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, H. Turner, Nonmonotonic causal theories, Artificial Intelligence 153 (1–2) (2004) 49–104.
[29] H. Kautz, B. Selman, Planning as satisfiability, in: B. Neumann (Ed.), Proc. of 10th European Conference on Artificial Intelligence (ECAI), IOS Press, 1992, pp. 359–363.
[30] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison–Wesley, 2002.
[31] R. Cavada, A. Cimatti, C.A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev, Nusmv 2.5 user manual, available at http://nusmv.fbk.eu, 2002.
[32] E. Pednault, Formulating multi-agents, dynamic world problems in the classical planning framework, in: M. Georgeff, A. Lansky (Eds.), Proc. of Reasoning about Actions and Plans, Morgan Kaufmann, 1987, pp. 47–82.
[33] R. Fikes, N.J. Nilsson, Strips: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (3–4) (1971) 189–208.
[34] C. Baral, M. Gelfond, Reasoning about effects of concurrent actions, J. Log. Program. 31 (1–3) (1997) 85–117.
[35] N. McCain, H. Turner, Causal theories of action and change, in: Proc. of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997), AAAI Press, 1997, pp. 460–465.
[36] J. Lee, V. Lifschitz, Describing additive fluents in action language $\mathcal{C}+$, in: G. Gottlob, T. Walsh (Eds.), Proc. of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03), Morgan Kaufmann, 2003, pp. 1079–1084.
[37] E. Erdem, A. Gabaldon, Representing action domains with numeric-valued fluents, in: M. Fisher, W. van der Hoek, B. Konev, A. Lisitsa (Eds.), Proc. of 10th European Conference on Logics in Artificial Intelligence (JELIA 2006), in: Lecture Notes in Comput. Sci., vol. 4160, Springer, 2006, pp. 151–163.
[38] V. Lifschitz, W. Ren, A modular action description language, in: Proc. of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006), AAAI Press, 2006.
[39] V. Akman, S.T. Erdogan, J. Lee, V. Lifschitz, H. Turner, Representing the zoo world and the traffic world in the language of the causal calculator, Artificial Intelligence 153 (1–2) (2004) 105–140.
[40] C. Castellini, E. Giunchiglia, A. Tacchella, Sat-based planning in complex domains: Concurrency, constraints and nondeterminism, Artificial Intelligence 147 (1–2) (2003) 85–117.
[41] C. Baral, T. Son, E. Pontelli, Modeling multi-agent domains in an action languages: An empirical study using $\mathcal{C}$, in: E. Erdem, F. Lin, T. Schaub (Eds.), Proc. of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), in: Lecture Notes in Comput. Sci., vol. 5753, Springer, 2009, pp. 409–415.
[42] O. Caldiran, K. Haspalamutgil, A. Ok, C. Palaz, E. Erdem, V. Patoglu, Bridging the gap between high-level reasoning and low-level control, in: E. Erdem, F. Lin, T. Schaub (Eds.), Proc. of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), in: Lecture Notes in Comput. Sci., vol. 5753, Springer, 2009, pp. 342–354.
[43] T. Eiter, E. Erdem, M. Fink, J. Senko, Comparing action descriptions based on semantic preferences, Ann. Math. Artif. Intell. 50 (3–4) (2007) 273–304.
[44] T. Eiter, E. Erdem, M. Fink, J. Senko, Updating action domain descriptions, in: L.P. Kaelbling, A. Saffiotti (Eds.), Proc. of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005), Professional Book Center, 2005, pp. 418–423.
[45] H. Davulcu, M. Kifer, C.R. Ramakrishnan, I.V. Ramakrishnan, Logic based modeling and analysis of workflows, in: Proc. of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1998), ACM Press, 1998, pp. 25–33.
[46] A. Chirichiello, Two formal approaches for web services: Process algebras & action languages, PhD thesis, “Sapienza” University of Roma, 2008.
[47] P. Koksal, N.K. Cicekli, I.H. Toroslu, Specification of workflow processes using the action description language $\mathcal{C}$, in: AAAI Spring 2001 Symposium Series: Answer Set Programming, 2001, pp. 103–109.
[48] L.M. de Moura, S. Owre, H. Rueß, J.M. Rushby, N. Shankar, M. Sorea, A. Tiwari, Sal 2, in: R. Alur, D. Peled (Eds.), Proc. of 16th International Conference on Computer Aided Verification (CAV 2004), in: Lecture Notes in Comput. Sci., vol. 3114, Springer, 2004, pp. 496–500.
[49] G.J. Holzmann, The model checker SPIN, Software Eng. 23 (5) (1997) 279–295.
[50] K. Knorr, H. Weidner, Analyzing separation of duties in Petri net workflows, in: V.I. Gorodetski, V.A. Skormin, L.J. Popyack (Eds.), International Workshop on Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Security (MMM-ACNS 2001), in: Lecture Notes in Comput. Sci., vol. 2052, Springer, 2001, pp. 102–114.
[51] H. Rakkay, H. Boucheneb, Security analysis of role based access control models using Colored Petri nets and CPNtools, Trans. Comput. Sci. (4) 5430 (2009) 149–176, special issue on security computing.
[52] W. Arsac, L. Compagna, G. Pellegrino, S.E. Ponta, Security validation of business processes via model-checking, in: Proc. of the Third International Symposium on Engineering Secure Software and Systems (ESSoS 2011), in: Lecture Notes in Comput. Sci., vol. 6542, Springer, 2011, pp. 29–42.
[53] S. Karch, L. Heilig, SAP NetWeaver, 1st edition, Galileo Press, 2004, http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+397693370&sourceid=fbw_bibsonomy.
[54] P.A. Bonatti, D. Olmedilla, Rule-based policy representation and reasoning for the semantic web, in: G. Antoniou, U. Aßmann, C. Baroglio, S. Decker, N. Henze, P.-L. Patranjan, R. Tolksdorf (Eds.), Reasoning Web, Third International Summer School 2007, Tutorial Lectures, in: Lecture Notes in Comput. Sci., vol. 4636, Springer, 2007, pp. 240–268.
[55] G. Antoniou, M. Baldoni, P.A. Bonatti, W. Nejdl, D. Olmedilla, Rule-based policy specification, in: T. Yu, S. Jajodia (Eds.), Secure Data Management in Decentralized Systems, in: Adv. Informat. Secur., vol. 33, Springer, 2007, pp. 169–216.
[56] P.A. Bonatti, J.L.D. Coi, D. Olmedilla, L. Sauro, Policy-driven negotiations and explanations: Exploiting logic-programming for trust management, privacy & security, in: Proc. of 24th International Conference on Logic Programming (ICLP 2008), in: Lecture Notes in Comput. Sci., vol. 5366, Springer, 2008, pp. 779–784.
[57] P.A. Bonatti, J.L.D. Coi, D. Olmedilla, L. Sauro, Rule-based policy representations and reasoning, in: Semantic Techniques for the Web: The REWERSE Perspective, in: Lecture Notes in Comput. Sci., vol. 5500, Springer, 2009, pp. 201–232.